Towards a Compositional Approach to Model Transformation for Software Development

Soichiro Hidaka
hidaka@nii.ac.jp

Zhenjiang Hu
hu@nii.ac.jp

Hiroyuki Kato
kato@nii.ac.jp

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

Keisuke Nakano
ksk@cs.uec.ac.jp

The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi, Tokyo 182-8585, Japan

ABSTRACT

Model transformation plays an important role in model-driven software development that aims to introduce significant efficiencies and rigor to the theory and practice of software development. Although models may have different notations and representations, they are basically graphs, and model transformations are thus nothing but graph transformations. Despite a large amount of theoretical work and a lot of experience with research prototypes on graph-based model transformations, it remains an open issue how to compose model transformations. In this paper, we report our first attempt at a compositional framework for graph-based model transformations using the graph querying language UnQL. The main idea of UnQL is that graph queries are fully captured by structural recursion that is suitable for efficient composition. We show that the idea can be applied to graph-based model transformations. We have implemented a prototype of the framework and tested it with several nontrivial examples. Our new framework supports systematic development of model transformation "in the large" with the advantage that it can automatically remove inefficiencies arising from their composition.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*software process models*; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages*; E.1 [**Data Structures**]: Graphs and Networks

General Terms

Management, Design, Languages

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

1. INTRODUCTION

Model transformation plays an important role in model-driven software development [11] that aims to introduce significant efficiencies and rigor to the theory and practice of software development. The specification, implementation, and execution of model transformation are the critical parts in model transformation [8]. A model transformation is considered just as a graph transformation since models are basically graphs, even though each model may have different notations and representations. This has led to the so-called *graph-based approach* [10, 14] to model transformations that is largely based on theoretical work in graph transformations [1, 7, 21].

Although the graph-based approach is powerful, being based on a large amount of theoretical work and experience with research prototypes, it remains a challenge to use it to develop model transformation in the large, which requires a composition mechanism with high modularity [8]. A recent survey paper [10] stated:

> *Open issues for all graph transformation approaches are elaborated concepts to compose transformations...*

It is true, from a practical point of view, that model composition would be necessary if one wants to chain and combine model transformations to produce new and more powerful transformations. To bridge large abstraction gaps between two models, it is often much easier to generate intermediate models rather than go straight to the target model. This would make model transformation more modular and maintainable. However, the graph-based approach lacks a good support for synthesizing an efficient model transformation from multiple transformations. Straightforward implementation of their sequential composition is generally inefficient because it constructs superfluous intermediate models.

In this paper, we report our first attempt to establish a compositional framework for graph-based model transformations, which supports concise specification of model transformation with the advantage that it can simplify and improve the efficiency of model transformation implementation and execution. This work was greatly inspired by the compositional graph querying language UnQL [6], which has been intensively studied in the database community. The key idea of UnQL is that all graph queries are fully captured by structural recursion that is suitable for efficient composition. We show that this idea can also be adapted to structure graph transfor-

mations to gain efficient composition. Our main contributions are twofold.

First, we propose a compositional framework for graph-based model transformations using the UnQL graph querying language. We designed a graph transformation language, UnQL$^+$, which extends UnQL with three simple graph editing constructs to achieve efficiency and expressiveness. All model transformation described in UnQL$^+$ can be mapped to structural recursions that are suitable for efficient composition.

Second, we implemented a prototype of the new framework and tested it with several nontrivial examples. Our new framework supports systematic development of model transformation in the large with the advantage that it can automatically remove inefficiencies arising from this composition. We demonstrate, using the nontrivial model transformation from classes to a relational database management system, that a large model transformation can be systematically developed by gluing simpler model transformations together. Since all model transformations are represented by structural recursions or their composition, our system automatically eliminates inefficiencies arising from composition by using fusion optimization. The experimental results show promising speedups. Additionally, our system can validate input and output models against given metamodels with an efficient algorithm.

The organization of this paper is as follows. In Section 2, we review the graph query language UnQL and its model of graphs. Section 3 describes how UnQL and its extension is useful for systematic development of model transformations in a compositional manner using a typical but nontrivial model transformation, Class2-RDBMS. In Section 4, we explain the architecture of our compositional framework and its implementation. We discuss related work in Section 5 and conclude the paper in Section 6.

2. GRAPH QUERYING LANGUAGE UNQL

In this section, we briefly review the graph querying language, UnQL [6]. Our compositional framework for model transformations is based on UnQL. This language has a convenient select-where style surface syntax, which is translated into a core graph algebra called UnCAL that consists of a small number of basic constructors and operators. Its expressive power is FO(TC) (first order with transitive closure), and its complexity in answering an UnQL query is PTIME. We present the basic concepts of UnQL starting with graph representation and bisimulation in UnQL.

2.1 Graph Representation

Graphs in UnQL are edge-labelled; that is, all information is stored as labels on edges rather than on nodes (the labels on nodes have no particular meaning). They are rooted and directed cyclic graphs whose orders between outgoing edges of a node are insignificant. Every node may be marked with an input or output marker, which is called an input or output node, respectively. Input markers are used to select entry points of the graph, whereas output markers are used to glue output nodes with input nodes of a graph.

Figure 1 shows an example of a graph that represents a class model, which will be shown in Figure 2. The numbers in circled nodes of the graph are just identifiers that are added for explanation. All information in the class model including object names, attribute names, and attribute values, appears as labels of edges of the graph. A shared object of the class model (e.g., the class object named "Address") is represented by a shared node in the graph (e.g., node 34). Some edges may be labelled with ϵ, which works like an ϵ transition in automata theory in that it identifies its source with its destination. They are used in establishing connections between nodes.

Graph bisimulation defines value equalities between graph instances. Intuitively, when graphs G_1 and G_2 are bisimilar, then every node x_1 in G_1 has a counterpart x_2 in G_2, and if there is an edge from x_1 to y_1, then there is a corresponding edge from x_2 to y_2. The UnQL data model extends graph bisimulation by (1) requiring equalities between labels, (2) allowing insertion of one or more consecutive ϵ edges between a normal edge and target node (y_1 or y_2 above), (3) requiring correspondence between input nodes in G_1 and G_2, and (4) requiring correspondence between output markers of corresponding nodes (output markers may be associated with a node other than corresponding nodes, provided that the marker is associated with nodes that can be reached by traversing ϵ edges).

The notion of extended bisimulation is useful because it allows variation in representing semantically equivalent graphs. It is surprising that a graph transformation defined in UnQL preserves bisimilarity [6] even though evaluation orders and strategies generally introduce divergence in results. If two graphs G_1 and G_2 are bisimilar, $f(G_1)$ and $f(G_2)$ are bisimilar for any transformation f in UnQL.

2.2 UnQL

UnQL, like other query languages, has a convenient select-where structure for extracting information from a graph. We omit the formal definition of the language syntax, which can be found in [6, 15]. We simply give some examples in this paper.

select-where construct. The following query `Q1` extracts all primitive data types from the database (denoted `$db` in the query) in Figure 1.

```
(* Q1 *)
select $T where
 {association:{dest:
   {class:{attrs:
     {attribute:{type:
       {primitiveDataType.name:$T}}}}}}} in $db
```

In `where`-clause, we can describe pattern matching as a condition which has the form of $pattern$ `in` $variable$, where $pattern$ is tree-structured as `{`$label$`:`$pattern'$`}` with $label$ and $pattern'$ which match an edge from its root and a graph following the edge, respectively. Unlike the original UnQL [6], variables are represented by `$`-prefixed symbols. We can use also regular path expressions for pattern matching at the left-hand side of the colon in the `where` clause.

Structural recursion in UnQL. Structural recursion plays a very important role in UnQL. Not only can it be used to describe many useful queries, but also any queries in UnQL can be described in terms of structural recursion.

Structural recursive function f in UnQL is a simple mutually recursive computation scheme, which satisfies the following two equations, $f\ \{\} = \{\}$ and $f\ (t_1 \cup t_2) = f(t_1) \cup f(t_2)$ for any graphs t_1 and t_2, where $\{\}$ stands for a graph consisting of an empty node. Additionally, it guarantees that any return value of functions should not be fed to another function. This simplicity allows manipulability of structural recursion, which is a combinator that is similar to the higher-order function map in functional programming languages. Whereas a map (on lists) is applied recursively to tail lists, structural recursion is applied (vertically) to nodes, as well as (horizontally) to edges.

As a simple use of structural recursion, the following query `Q2` replaces all labels `name` under `primitiveDataType` in Figure 1 with `typeName`. Due to the two equations above, definitions for horizontal recursion are always omitted.

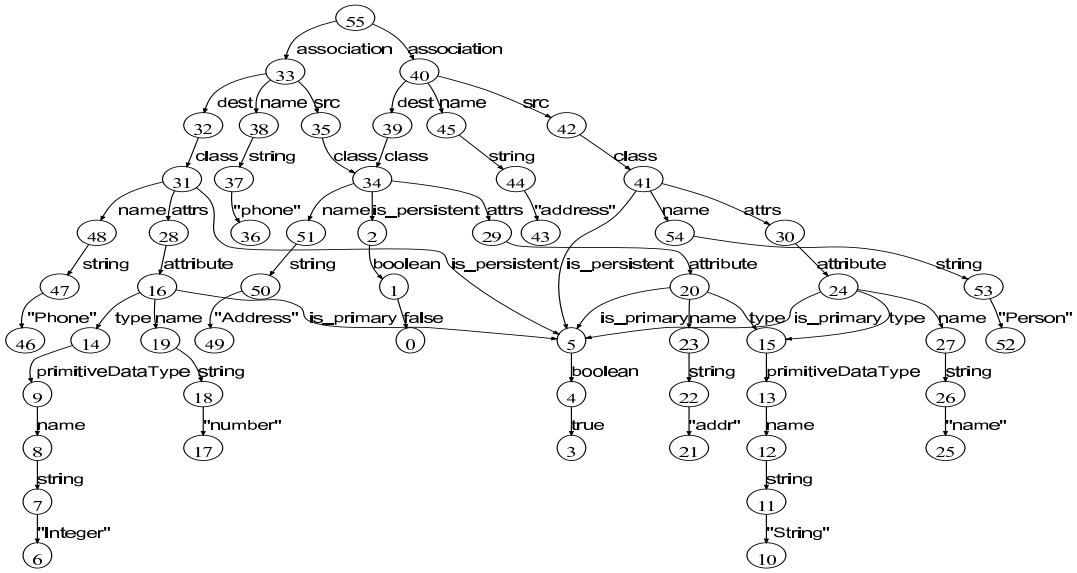


Figure 1: Class model represented by an edge-labelled graph

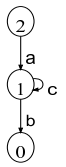
```
 (* Q2 *)
select
 letrec sfun f1 ({primitiveDataType:$T})
                        = {primitiveDataType:g1($T)}
          | f1 ({$L:$T}) = {$L:f1($T)}
 and    sfun g1 ({name:$T})  = {typeName:g1($T)}
          | g1 ({$L:$T})     = {$L:g1($T)}
 in f1($db)
```

The function `f1` takes the input graph `$db` and matches all edges from its root with either `{primitiveDataType:$T}` or `{$L:$T}`. According to the definition of `f1`, it processes deeper subgraphs recursively. Since the subgraphs do not include the matched label, the recursion always terminates.

2.3 UnCAL: A Graph Algebra

While UnQL is an interface language for users to write queries, UnCAL is its core language for internal implementation. UnCAL has a set of constructors and operators, by which arbitrary graphs can be represented. In addition to tree constructors, graph concatenation and cycle operator, together with input and output markers, form cycles and confluences by gluing nodes marked with identical markers together. Complete syntax and brief semantics of UnCAL expressions are depicted in [15].

Contrary to the appearance of tree constructor $\{\}$ and $\cup$, its semantics of unification is different from those of sets. In UnCAL, although value equality is explicitly defined, duplicate eliminations do not take place. The graph shown in the right figure is represented by the following UnCAL expression




```
&z2@cycle((&z2 := {a:&z1},
          &z1 := {c:&z1,b:&z0},
          &z0 := {}))
```

where `&z0`, `&z1`, and `&z2` correspond to the three nodes, 0, 1, and 2, respectively, and `a`, `b`, and `c` correspond to the three edges.

3. MODEL TRANSFORMATION IN UnQL$^+$

This section explains one of our most important contributions. We show how to extend the graph querying language UnQL [6] to UnQL$^+$ with three useful editing operations, and demonstrate how it can be used for systematic development of model transformations with the example of the transformation from a class model to a relational database management system model.

3.1 UnQL$^+$

UnQL is suitable for graph querying but not for graph transformation. For example, an UnQL query

```
select $Class where
   {_*.class: $Class} in $db,
   {is_persistent: true} in $Class
```

only extracts all of the persistent classes in Figure 1. In graph transformation, we often want to delete a subgraph, extend a subgraph with some new information, or replace a subgraph with a new one. It is onerous to describe these kinds of graph transformations in UnQL because we need to preserve the context by copying some structures in the input. UnQL$^+$ is an extension of UnQL with three editing constructs that can support direct specification of these graph transformations (model transformations).

The deletion construct, `delete ... where ...`, is introduced to describe deletion of part of the graph. Consider the class graph in Figure 1, and suppose we want to eliminate all the names of association. This can be described by

```
delete $AssocName
where {association.name: $AssocName} in $db
```

where the subgraph matched with `$AssocName` will be deleted from its original graph. In contrast, the following transformation extracts the association names as a result.

```
select {result: $AssocName}
where {association.name: $AssocName} in $db
```

Therefore, we may consider the `delete` as the dual of the `select`.

The extension construct, `extend ... where ...`, is introduced to extend a graph with another graph. For example, we write the following transformation to add `date` information to each `association`.

```
extend $AG with {date:"2008/8/4"}
where {association: $AG} in $db
```

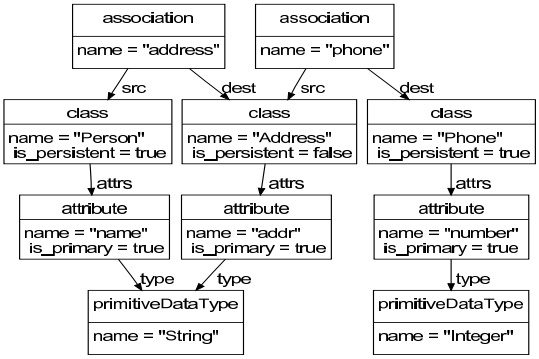


Figure 2: Class model

The replacement construct, `replace ... where ...`, is introduced to replace a subgraph with a new subgraph. For example, the transformation of replacing the edge label `dest` by `tgt` can be specified as follows.

```
replace $G by {tgt:$G1}
where {association: $G} in $db,
      {dest: $G1} in $G
```

It is worth noting that as will be seen in Section 4.2, these new editing constructs can be mapped to structural recursions, and thus, all the advantages of UnQL, including the compositional property, are preserved.

3.2 Example: Class2RDBMS

As a nontrivial example, we consider the model transformation, Class2RDBMS, a simplified version of the well known "Class to RDBMS" transformation. It was proposed as a common example to all the participants of the International Workshop on Model Transformations in Practice 2005 [3], whose purpose was to compare and contrast various approaches to model transformation. We explain two models, Class and RDBMS, and the requirement for a model transformation from Class to RDBMS, before showing how it can be systematically developed in UnQL$^+$.

Class Model. A class model consists of classes and directed associations. A class is indicated as persistent or non-persistent. It consists of one or more attributes, at least one of which must be marked as constituting the classes' primary key. An attribute type is of a primitive data type (e.g. String, Integer). Associations are used to associate two classes. Figure 2 shows a class model, which consists of three classes and two directed associations. This class model is represented by the graph in Figure 1, where all information is stored on edges instead of nodes.

RDBMS Model. An RDBMS model consists of one or more tables. A table consists of one or more columns. One or more of these columns will be included in the pkey slot of a table, denoting that the column forms part of the table's primary key slot. A table may also contain zero or more foreign keys. Each foreign key refers to the particular table it identifies, and denotes one or more columns in the table as being part of the foreign key. Figure 3 shows an RDBMS model that has two tables.

Specification of Class Models to RDBMS Models. We recap the informal specification [3] of Class2RDBMS, the model transformation from class models to RDBMS models. A persistent class is mapped to a table and all its attributes or associations to columns in this table. If the type of attribute is primary, a primary key from the table to the column is established. If the type of attribute or association is another persistent class, a foreign key to the corresponding table is established. If class hierarchies are transformed, only the topmost classes are mapped to tables. Additional attributes and associations of subclasses result in additional columns of the top-most classes. Non-persistent classes are not mapped to tables; however, one of the main requirements for the transformation being considered is to preserve all the information in the class diagram. That means attributes and associations of non-persistent classes are distributed over those tables stemming from persistent classes that access non-persistent classes. This model transformation is not trivial. We show below how to systematically develop it in our compositional framework.

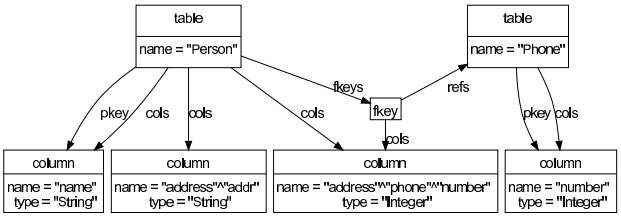


Figure 3: RDBMS model

Class2RDBMS in UnQL$^+$. The compositional framework of UnQL$^+$ allows us to develop bigger model transformations by gluing together smaller transformations via intermediate models, without worrying about inefficiency due to the intermediate models. This is because unnecessary intermediate models will be removed automatically by our system. The entire transformation of Class2-RDBMS in UnQL$^+$ is given in [15]. Let us explain how it is systematically developed.

Recall the specification of Class2RDBMS, where we want to create tables (independent tables or tables pointed by foreign keys) from a class diagram, where each table should have a name and a sequence of columns, some of which are pointed by primary or foreign keys. This leads to the following top-level transformation.

```
select {table: {name: {$Name: {}}} U
               $MakePKeyCol U
               $MakeGenCol U
               $MakeFKeyCol,
        table: $MakeFKeyTable}
where ...
```

For the input class model `$db`, as a preprocessing step, we replace all primitive data types in `$db` with their names and get `$db'` (because only the type names are used in the tables.)

```
$db' in
 (replace $PrimDT by $Name
  where {_*.type: $PrimDT} in $db,
        {primitiveDataType.name: $Name} in $PrimDT
```

Now, to create columns of a table, we need to gather all the information on classes that are directly or indirectly associated with the source persistent class. This means we need to create an intermediate model `$ChainDB`, in which indirectly associated classes are directly associated.

```
$ChainDB in
(select
  ...
 where {association:
```

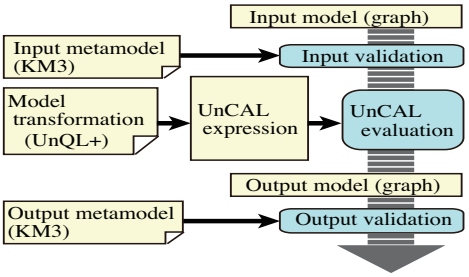


Figure 4: Overview of our system

```
  {name: $N1, src: $C11, dest: $C12}} in $db',
{association:
  {name: $N2, src: $C21, dest: $C22}} in $db',
{name: {$Name12: $Any}} in $C12,
{name: {$Name21: $Any}} in $C21,
$Name12 = $Name21)
```

We look for two associations in which one's destination is the other's source, and then add a link edge from the source to the indirect destination Note that we do not need to worry about the relationship between the new and the old models, and this new model is just for intermediate use.

With `$ChainDB`, it is easy to "query" the graph to extract information from each top (source) class that is persistent for creating a table later.

```
{group: {src: {name: {$Name:  $Any},
               is_persistent: true,
               attrs: $As},
        chains: $Chains}} in $ChainDB
```

From the information obtained, we can create a primary key for the table by querying the data from the graphs and add two new edges `pkey` and `cols`.

```
$MakePKeyCol in
  (select {pkey: $Col, cols: $Col}
   where {attribute: ...} in $As,
         $Primary = true,
         $Col in {column: ...})
```

Note that `$Col` is another shared intermediate model (graph), which appears twice in the `select` part. We omit an explanation of definitions for creating other columns and foreign keys, which are very similar.

As seen from this example, UnQL$^+$ enables us to productively develop model transformations in a compositional manner (we can glue results with unison operator `U` or sequentially apply simpler model transformations with some intermediate models). This good result is not surprising, because the usefulness of composition has been widely known in the development of program transformation.

The execution of this model transformation on the class graph in Figure 1 yields a graph corresponding the table diagram in Figure 3.

4. COMPOSITIONAL MODEL TRANSFORMATION SYSTEM

This section describes our system implemented in Objective Caml. Figure 4 shows an overview of our model transformation system. An input model represented by a graph is validated against a given input metamodel described in Kernel MetaMetaModel (KM3) [2]. The validated graph is transformed by a given program described in UnQL$^+$. The transformation is performed by translating the program into an UnCAL expression, which is a structural recursion

```
package Class {
  datatype String;
  datatype Boolean;
  abstract class NamedElt {
    attribute name : String;
  }
  class Association extends NamedElt {
    reference src : Class;
    reference dest : Class;
  }
  class Class extends NamedElt {
    attribute is_persistent : Boolean;
    reference attrs [1-*] : Attribute;
  }
  class Attribute extends NamedElt {
    attribute is_primary : Boolean;
    reference type : PrimitiveDataType;
  }
  class PrimitiveDataType extends NamedElt {}
}
```

Figure 5: KM3 metamodel for classes

over an input graph, in a way similar to the original UnQL implementation [6]. The output graph of the transformation is validated against a given output metamodel in KM3. We also show how our model compositions work efficiently. The interest readers can find and execute some transformation examples in our demonstration Web pages at `http://research.nii.ac.jp/~hidaka/big/`.

4.1 Metamodel and Validation

Our system validates both input and output models represented by graphs against given metamodels of them. We employ Kernel MetaMetaModel (KM3) to describe metamodels because it has been used as a metametamodel in actual software development and is more formally defined than other metametamodels. A KM3 metamodel prescribes which sets of nodes must be referred to by a node by a regular expression. See [2] for details on the specification of KM3.

Figure 5 shows an example of a KM3 metamodel for classes, each of which is an input for the model transformation introduced in Section 3.2. The metamodel consists of four classes, `Association`, `Class`, `Attribute`, and `PrimitiveDataType`. A class has some *features*, either *reference* or *attribute*. Every feature has a type, either class or data type. Since all of them inherit their super class `NamedElt`, they have an attribute `name`, which is `String`. For example, the `Association` class has two references `src` and `dest` which are `Class`.

We validate a graph by associating each edge in them with the name of a class or a feature in a given KM3 metamodel. The validation of a graph starts with its root node. Every edge from the root node is associated with a class name. For example, an edge to node 33 in Figure 1 is associated with a class `Association`. Next we associate edges that follow it with feature names of the class. For example, an edge to node 32 is associated with the reference `dest`. We visit all edges repeatedly in this way and record matching information. This process is completed in linear time with regard to the size of the graph.

4.2 Mapping to the Core Language

UnQL$^+$ provides a friendly interface for users to describe model transformations. For efficient implementation, UnQL$^+$ can be transformed to the core language UnCAL, where structural recursion plays an important role in supporting efficient composition of model transformations. UnQL$^+$ is mapped to UnCAL in a similar way to

that found in [6] except for the editing constructs. We show how to eliminate editing constructs in UnQL$^+$ to enable mapping from UnQL$^+$ to UnCAL.

First, deletion or extension of a subgraph can be expressed by the `replace` construct based on the following two rules.

```
delete $G where ...
  => replace $G by {} where ...

extend $G with $G1 where ...
  => replace $G by $G U $G1 where  ...
```

Second, the `replace` construct can be eliminated using the `select` construct and structural recursions. After simplification of the `where` clause, the `where` clause becomes a sequence of boolean conditions bc of relation expressions r such as `$A=5`, simple pattern-in boolean expressions pi such as $\{pat$`:$G`$\}$ `in` $template$ with a pattern pat and variable `$G`, or simple binding expressions bd such as `$G in` $template$ with a variable `$G`. Thus, the general form of an expression using `replace` is

`replace $G1 by $G2`
`where` $bc_1, \ldots, bc_{k-1}, \{pat$`:$G1`$\}$ `in $D`$, bc_{k+1}, \ldots, bc_n$

where $bc_1, \ldots, bc_{k-1}$ are either relation expressions or pattern-in boolean expressions. Among others, two important rules are as follows to reduce the number of boolean conditions in the where clause. In the following rules, like [6], we use $rest$ for a syntactic meta-variable which stands for the remaining clauses in the `where` component. Note that the first element in $rest$ is restricted to pi or bd unless $rest$ is empty. The first rule is to deal with the case where the first pattern-in boolean expression (in the `where` clause) does not match the graph to be replaced, i.e., `$G1 != $G3`.

```
replace $G1 by $G2 where {l:$G3} in $D,r1,...,rm, rest
=>
let sfun h1({$L:$G3}) =
      if $L=l and r1 and ... and rm then
         {$L:(replace $G1 by $G2 where rest)}
      else {$L:$G3}
in h1($D)
```

The second rule, on the other hand, is to deal with the case where the first boolean condition matches the graph to be replaced.

```
replace $G1 by $G2 where {l:$G1} in $D, r1,...,rm, rest
=>
let sfun h1({$L:$G1}) =
      if $L=l and r1 and ... and rm then
         letval $G1' = select {l:{}} where rest in
         letval $G2' = select $G2 where rest in
         if isEmpty($G1') then {L:$G1} else {L:$G2'}
      else {L:$G1}
in h1($D)
```

To see how these rules work, consider the following expression with `replace`, which is to replace the association name `"phone"` with `"assoc_phone"`.

```
replace $Name by $Name'
where {association:$Assoc} in $db,
      {name:$Name} in $Assoc,
      {string:$Na} in $Name,
      {$N:{}} in $Na,
      $N = "phone",
      $Name' in {string:{"assoc_"^$N:{}}}
```

With the two rules, we can successfully desugar it to the following, where `replace` is removed.

```
let sfun h1({$L:$Assoc}) =
   if $L=association then
     let sfun h2({$L:$Name}) =
       if $L=name then
         letval
         $G1' = (select {name:{}}
                 where
                 {string:$Na} in $Name,
                 {$N:{}} in $Na,
                 $N = "phone",
                 $Name' in {string:{"assoc_"^$N:{}}})
         in
         letval
         $G2' = (select $G2
                 where
                 {string:$Na} in $Name,
                 {$N:{}} in $Na,
                 $N = "phone",
                 $Name' in {string:{"assoc"^$N:{}}})
         in
         if isEmpty($G1') then {$L:$Name} else {$L:$G2'}
       else {$L:$Name}
     in h2($Assoc)
  else {$L:$Assoc}
 in h1($db)
```

4.3 Interpretation of the Core Language

Buneman et al.'s UnQL paper [6] provides two evaluation strategies that are proved to be equivalent: *bulk* semantics and *recursive* semantics. The latter is intuitive in that applications of body expression (e_1 in $rec(e_1)(e_2)$) take place in a top-down fashion. Revisiting of nodes caused by cycles can be correctly handled by memoization. The former deals with possible cycles by applying e_1 once for every edge in an input graph and connecting them together using Skolem functions on markers and nodes. Our system implemented in Objective Caml uses an algebraic data type for the Skolem function and a tree-structured set library for the first-order formula on nodes and edges. It is fairly straightforward and efficient in both recursive semantics and bulk semantics.

4.4 Model Composition

We identify two forms of model composition. The first one is a pair of consecutive transformations, T_1 and T_2, where the output model of T_1 is the input model of T_2: $\mathcal{M}' = (T_2 \circ T_1)(\mathcal{M}) = T_2(T_1(\mathcal{M}))$. The second one is a pair of transformations, T_1 and T_2, that share an identical input model: $(\mathcal{M}_1, \mathcal{M}_2) = (T_1 \triangle T_2)(\mathcal{M}) \stackrel{\text{def}}{=} (T_1(\mathcal{M}), T_2(\mathcal{M}))$. In the first composition, an intermediate result can be eliminated by the fusion technique, while in the second composition, a duplicate traversal of the input model can be unified by the tupling technique. Our system provides automatic fusion for the first composition.

In our framework, consecutive model transformations are translated into a composition of structural recursions in UnCAL. Hence, we can directly apply fusion transformation for the UnCAL proposed in [6]. As a very simple case, consider the following scenario (borrowed from [6]): first apply `Q2` (in Section 2) to the model in Figure 1, and then retrieve all `name`s by the following query `Q3`.

```
(* Q3 *)
 select
  letrec sfun f2 ({name:$T}) = {name:g2($T)}
           |  f2 ({$L:$T})   = f2($T)
   and   sfun g2 ({$L:$T})   = {$L:g2($T)}
  in f2($db)
```

This compositional query would look like the following query `Q4`, by which our desugaring module produces an UnCAL query `Q5`. Our UnCAL rewriter translates it into `Q6` (with simple rewriting by hand for readability), where two recs in `Q5` are fused into one.

```
(* Q4 *)
```

```
select
 letrec sfun f2 ({name:$T})  = {name:g2($T)}
          |  f2 ({$L:$T})    = f2($T)
 and    sfun g2 ({$L:$T})    = {$L:g2($T)}
 in
 letrec sfun f1 ({primitiveDataType:$T})
              = {primitiveDataType:g1($T)}
          |  f1 ({$L:$T})   = {$L:f1($T)}
 and    sfun g1 ({name:$T}) = {typeName:g1($T)}
          |  g1 ({$L:$T})   = {$L:g1($T)}
 in f2(f1($db))
```

```
(* Q5 *)
 &z1@rec(\ ($L,$T).
   if $L = "name"
   then (&z1 := {"name": &z2},
         &z2 := {"name": &z2})
   else (&z1 := &z1, &z2 := {$L: &z2}))
   (&z1@rec(\ ($L,$T).
     if $L = "name"
     then (&z1 := {"name": &z1},
           &z2 := {"typeName": &z2})
     else if $L = "primitiveDataType"
       then (&z1 := {"primitiveDataType": &z2},
             &z2 := {"primitiveDataType": &z2})
       else (&z1 := {$L: &z1}, &z2 := {$L: &z2}))
       ($db))
```

```
(* Q6 *)
 &z1@(&z2 := &z1&z2, &z1 := &z1&z1)@
  rec(\ ($Sa1,$T).
   if $Sa1="name"
   then (&z1&z1 := {"name": &z1&z2},
         &z1&z2 := {"name": &z1&z2},
         &z2&z1 := &z2&z1,
         &z2&z2 := {"typeName": &z2&z2})
   else if $Sa1 = "primitiveDataType"
     then (&z1&z1 := &z2&z1,
           &z1&z2 := {"primitiveDataType": &z2&z2},
           &z2&z1 := &z2&z1,
           &z2&z2 := {"primitiveDataType": &z2&z2}
     else (&z1 := llet $L = $Sa1 in
             if $L = "name"
             then (&z1 := {"name": &z1&z2},
                   &z2 := {"name": &z1&z2})
             else (&z1 := &z1&z1,
                   &z2 := {$L: &z1&z2}),
           &z2 := llet $L = $Sa1 in
             if $L = "name"
             then (&z1 := {"name": &z2&z2},
                   &z2 := {"name": &z2&z2})
             else (&z1 := &z2&z1,
                   &z2 := {$L: &z2&z2})
            ))($db)
```

The table below shows the efficiency gained from the above fusion. The experiment was conducted on a 1.5 GHz quad Xeon SMP machine running Linux kernel 2.4.20. An approximate three- to five-fold speed-up was confirmed.

evaluation strategy of *rec*	before fusion	after fusion	speed-up ratio
bulk	1.31 sec	0.25 sec	5.32
recursive	2.08 sec	0.73 sec	2.86

Our system employs the optimization rules presented in [6] to reduce the size of arguments of *rec*. In addition, we introduce the following simplification rules for other constructs to optimize graph transformation.

$$\&x := (\&z := e) \downarrow \&x.\&z := e$$
$$\&x := (e_1 \oplus e_2) \downarrow (\&x := e_1) \oplus (\&x := e_2)$$

$$e \cup \{\} \downarrow e \qquad \{\} \cup e \downarrow e$$
$$e \oplus () \downarrow e \qquad () \oplus e \downarrow e \qquad () \mathbin{@} e \downarrow ()$$

$cycle(e) \downarrow e$ if input and output markers are disjoint in e.

where we follow the notation of graph constructs in [6]. The operator $\oplus$ constructs disjoint union of two graphs while () denotes an empty graph, thus, the expressions like $\{\&z_1 := g_1, \&z_2 := g_2, \ldots, \&z_n := g_n\}$, which we have already seen in this paper are the syntactic shorthands for $(\&z_1 := g_1) \oplus (\&z_2 := g_2) \oplus \ldots \oplus (\&z_n := g_n)$. The last rule requires that the sets of input and output markers be inferred for a given UnCAL expression. Our system is capable of doing this estimation at compile time in a way similar to that in [5]. There may be other rules applicable. Exploring these rules will be part of our future work.

5. RELATED WORK

Our work is very closely related to research on model transformation based on graph transformation in the software engineering community, as well as to research on graph querying in the database community.

In the software engineering community, graph transformation has been widely used for expressing model transformations [10, 19, 16].

AGG [23, 9] is a well-known rule-based visual tool that supports an algebraic approach to graph transformation. AGG supports typed (attributed) graph transformations including type inheritance and multiplicities. Rule application can contain a non-deterministic choice of rules that may be controlled by rule layers. Different from our approach, AGG does not have a clear separation between the source and target graphs. It is not straightforward to compose/write multi-staged transformations in AGG.

Triple Graph Grammars (TGG) [17, 12] were proposed as an extension of Pratt's pair grammar approach [20], which aims at the declarative specification of model-to-model integration rules. TGGs consist of a schema and a set of graph rewriting rules, and they explicitly maintain the correspondence of two graphs by means of correspondence links. These correspondence links play the role of traceability links that map elements of one graph to elements of the other graph and vice versa. With TGG, one has to explicitly describe correspondence between the source and target models, which is difficult if the transformation is complex and the intermediate models are required during the transformation.

Neither AGG nor TGG has strict control over application of elementary algebraic graph transformation rules. To increase usability and efficiency of graph transformation, a variety of control concepts for rule and match selection have been considered in many graph transformation approaches such as VIATRA [4] and VMTS [18], where graph transformations are controlled with recursive graph patterns. Unlike AGG and TGG, graph transformation rules are guaranteed to be executable, which is the main conceptual difference. Since their recursive control structures can be very complicated, it remains unclear how to efficiently compose them. Our approach puts reasonable restrictions on the recursive structure so that it is not only powerful enough to specify various model transformations but also suitable for efficient composition.

On the other hand, in the database community, a lot of work has been done on language design and implementation for efficient graph querying [13, 22, 6]. Different from querying trees, issues on representation and equivalence of graphs are subtle and important to define the correctness of graph querying (as well as graph transformation), and the use of bisimulation and structural recursion in [6] leads to a very nice framework for both declarative and

efficient graph querying with high modularity and composability. This has motivated us to see if we can extend the framework from graph querying to graph transformation.

6. CONCLUSION

In this paper, we have reported our first attempt to design and implement a compositional framework for model transformations based on UnQL. Although UnQL is well known in the database community for its unique solution to the composition problem, no one, as far as we are aware, has recognized its usefulness in software development. We have shown that it is indeed useful and that the main theory and technique can be applied to solve the composition problem in model transformations.

We are currently working on extending this framework further to add "bidirectionality" to the compositional model transformation so that updates on the target model can be reflected in the source model. This would connect the interesting idea of bidirectional computation in both the programming language and software engineering communities.

7. ACKNOWLEDGEMENTS

We would like to thank Mary Fernandez from AT&T Labs Research, who kindly provided us with the SML source codes of an UnQL system, which helped us a lot in implementing our extended system in Objective Caml.

Our research was supported in part by the Grand-Challenging Project on "Linguistic Foundation for Bidirectional Model Transformation" from National Institute of Informatics, the National Natural Science Foundation of China under Grant No. 60528006, the Grant-in-Aid for Scientific Research (C) No. 20500043, and Encouragement of Young Scientists (B) of the Grant-in-Aid for Scientific Research No. 20700035.

8. REFERENCES

[1] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. In *704*, page 55. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 30 1995.

[2] ATLAS group. KM3: Kernel MetaMetaModel manual. `http://www.eclipse.org/gmt/atl/doc/`.

[3] J. Bezivin, B. Rumpe, and T. L. Schürr A. Model transformation in practice workshop announcement. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*. Springer-Verlag, 2005. `http://sosym.dcs.kcl.ac.uk/events/mtip/`.

[4] E. Börger, A. Gargantini, and E. Riccobene, editors. *Abstract State Machines, Advances in Theory and Practice, 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003, Proceedings*, volume 2589 of *LNCS*. Springer, 2003.

[5] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. Technical Report MS-CIS-96-21, University of Pennsylvania, 1996.

[6] P. Buneman, M. F. Fernandez, and D. Suciu. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal: Very Large Data Bases*, 9(1):76–110, 2000.

[7] A. Corradini and F. Gadducci. A 2-categorical presentation of term graph rewriting. In *Category Theory and Computer Science*, pages 87–105, 1997.

[8] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[9] H. Ehrig, K. Ehrig, G. Taentzer, J. de Lara, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In J. R. Cordy, R. Lämmel, and A. Winter, editors, *Transformation Techniques in Software Engineering*, volume 05161 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[10] K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, and S. Varró-Gyapay. Model transformation by graph transformation: A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*. Springer-Verlag, 2005.

[11] D. S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.

[12] H. Giese and R. Wagner. Incremental model synchronization with triple graph grammars. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Models '06: Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 543–557. Springer Verlag, October 2006.

[13] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs, 2002.

[14] L. Grunske, L. Geiger, and M. Lawley. A graphical specification of model transformations with triple graph grammars, 2005.

[15] S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Towards compostional approach to model transformations for software development. Technical Report GRACE-TR08-01, GRACE Center, National Institute of Informatics, Aug. 2008.

[16] F. Jouault and I. Kurtev. Transforming models with ATL. In *Proceedings of Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 128–138. Springer, 2006.

[17] A. Konigs and A. Schurr. Tool integration with triple graph grammars - a survey. *Electronic Notes in Theoretical Computer Science*, 148(1):113–150, February 2006.

[18] L. Lengyel, T. Levendovszky, G. Mezei, and H. Charaf. Model transformation with a visual control flow language. *International Journal of Computer Science (IJCS)*, 1(1):45–53, 2006.

[19] OMG. MOF QVT final adopted specification. `http://www.omg.org/docs/ptc/05-11-01.pdf`, 2005.

[20] T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *J. Comput. Syst. Sci.*, 5(6):560–595, 1971.

[21] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[22] L. Sheng, Z. M. Ozsoyoglu, and G. Ozsoyoglu. A graph query language and its query processing. In *ICDE*, pages 572–581, 1999.

[23] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *AGTIVE*, volume 3062 of *LNCS*, pages 446–453. Springer, 2003.